Integrating Independent Components with On-Demand Remodularization

Mira Mezini
Darmstadt University of Technology
D-64283 Darmstadt, Germany
mezini@informatik.tu-darmstadt.de

Klaus Ostermann
Siemens AG, CT SE 2
D-81730 Munich, Germany
Klaus.Ostermann@mchp.siemens.de

ABSTRACT

This paper proposes language concepts that facilitate the separation of an application into independent reusable building blocks and the integration of pre-build generic software components into applications that have been developed by third party vendors. A key element of our approach are *on-demand remodularizations*, meaning that the abstractions and vocabulary of an existing code base are translated into the vocabulary understood by a set of components that are connected by a common *collaboration interface*. This general concept allows us to mix-and-match remodularizations and components on demand.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Programming Techniques—*Reusable Software*; D.3.3 [**Software Engineering**]: Programming Languages—*Modules, packages*

General Terms

Design, Languages

Keywords

On-Demand Remodularization, Aspect-Oriented Programming, Collaboration-Based Decomposition

1. INTRODUCTION

This paper proposes language support for separating and capturing generic application logic that is useful in several places within one application domain or even across application domains. Hence the attribute 'generic'. In the terminology of standard component models such as CORBA, one would probably use terms such as vertical and horizontal facilities for the kind of generic application logic we target here.

Graph algorithms are a good match for the kind of the generic functionalty we mean, since they are used in almost any application domain. They are often even instantiated several times within the same application, whereby each instantiation might involve different units in the modular structure of the application in playing the roles of vertices and edges in the world of graph abstractions. For example, the university administration software in Fig. 1 can be viewed as a graph whose vertices are the courses and whose edges are gained by connecting any pair of courses that are taught by the same teacher or required by the same student year. This view on the module structure of the university software would be necessary on the demand of applying graph coloring[1] to compute an optimal assignment of time slots to courses. Other - eventually completely different or overlapping - remodularized views of the university software are needed on the demand of adding other features by either differently applying the coloring algorithm or by applying other graph algorithms, e.g., graph matching[2]. Tab. 1 presents different sample representations of the university administration example as a graph.

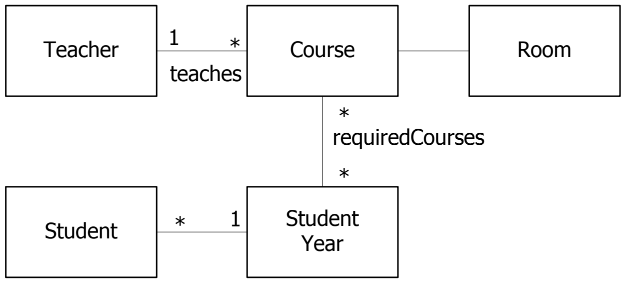

Figure 1: University Example

One can easily generalize from graph algorithms to generic application logic in other domains such as e.g., price calculation logic in the domain of web-based order systems, bonus calculation and administration logic in the domain of online travel agency software, etc. Now that we have illustrated the meaning of the term generic functionality, the message we want to convey is that appropriate language technol-

[1]A graph coloring is an assignment of colors to the vertices of a graph such that no two vertices that are connected by an edge have the same color. A minimum coloring is a coloring with a minimum number of colors.

[2]A matching is a subset $M \subseteq E$ of the graph's edges such that every vertex is connected to at most one edge from M. A maximum matching is matching with a maximum number of edges in M.

Graph	**Vertex**	**Edge (v1,v2)**
Course Collision	Courses	Teacher teaches both v1 and v2 or both v1 and v2 required by same student year
Student Contacts	Students	v1 and v2 visit a common course
Student knows Teacher	Students, Teachers	Student v1 visits a course by teacher v2
Teacher uses Room	Teachers, Rooms	Course with Teacher v1 and assigned room v2

Table 1: Possible Mappings from University Example to Graph

ogy should support a software development process in which such generic functionality as graph coloring or price calculation (a) are provided as 'off-the-shelf' components whose implementation is decoupled from any particular application, and (b) can be integrated a-posteriori into a multitude of existing software.

The requirement for independent implementation of generic functionality calls for appropriate module constructs for doing so. The requirement for a-posteriori integration implies that we need a remodularization of the existing software, however, without physically changing it. A physical change is not only undesirable but also frequently impossible. There is in fact no single physical change of the modular structure that would satisfy the needs of all generic functionality to be integrated, since they have in general quite different views of what the modular structure should be.

In the absence of appropriate language technology, the implementation of different graph algorithms will be scattered around several classes, offen dublicated, rendering the resulting software a nightmare to maintain and evolve. To use the terminology of aspect-oriented software develeopment comunity, graph algorithm implementations would crosscut the modular structure of the university software. Unfortunately, as argued in [2, 8, 17], current object-oriented languages are not very well equipped to cope with the subtle problems that occur when integrating independently developed components. Industrial component models such as EJB and CCM do not tackle this problem, either. With beans in the EJB model one can indeed 'write application logic once and run it in (almost) any server platform'. However, the integration of generic independently developed application logic into existing EJB software is not supported.

The problem does not only apply to the integration of components from third party vendors but also to the integration of reusable modules in general: those that capture different separated concerns of a system in any software engineering effort. The principles of *separation of concerns* [5], which manifests itself today in the form of *aspect-oriented programming* [13, 27, 1, 20], tells us that we should try to divide our software in smaller pieces that are as independent from each other as possible, in order to facilitate maintenance, understandability and reusability. However, as indicated in [11], the aspect-oriented programming community recognizes that still much has to be done for supporting flexible integration of crosscutting concerns.

The work presented in this paper aims at improving the state-of-the-art technologies targeted at the problem domain outlined so far. To support independent implementation of generic functionality, we introduce *collaboration interfaces* for declaring generic component types. Collaboration interfaces differ in two ways from standard interfaces as we know them, e.g., from Java.

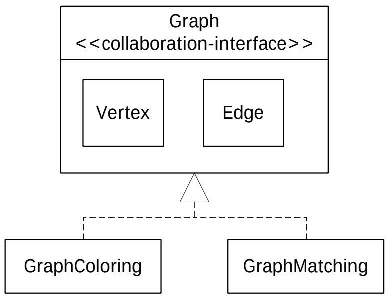

Figure 2: Graph collaboration

First, they can be nested, thereby allowing the bundling of several abstractions that together build up the concept world of a component type. For example, a component that provides algorithms on graphs articulates its world outlook - the structure and the requirements of a graph to which the algorithms could be applied - in the collaboration interface. Other components that also operate on graphs can refer to the same collaboration interface. This is schematically illustrated in Fig. 2. Second, in addition to expressing what a client can expect from an implementation of the interface - the *provided contract* -, collaboration interfaces also explicitly capture what interface implementations expect from potential client contexts in which they might be integrated. We say, they also make explicit the *expected contract*.

In order to support flexible a-posteriori integration of generic components into existing applications, we distinguish between *implementing* and *binding* a collaboration interface (CI). *Implementing a CI* means implementing its provided contract, while *binding a CI* means implementing its expected contract. Binding a collaboration interface is done with respect to a particular application into which the component gets integrated. We assume that the world of the particular application is, in general, very different from the component's world. Therefore, we provide language means to express how the abstractions of a base application should be translated to the vocabulary of a particular collaboration interface. We use the term *on-demand remodularization* for this translation process to indicate two important characteristics of our remodularization concept. First, remodularizations in our model are virtual, meaning that the base module structure is never changed physically; a remodularization rather defines a virtual view on top of the physical structure. Second, the remodularization specified for binding a collaboration interface `C` is effective only on the demand of executing functionality in `C`. In other words, the semantics of existing programs remains unchanged as long as the remodularization is not explicitly applied.

Please note that our use of the term is not identical to its use in the context of HyperJ [22], where it was originally introduced. We will explain the difference later.

An important insight that drove our approach to linguistic means to express bindings is that simple mappings from component abstractions to base classes ("In the collaboration C, class X plays the role R") are not sufficient. The base application does not necessarily have classes that do directly correspond to a role in a particular collaboration. For instance, there is no abstraction in the university software that directly corresponds to the edge abstraction in the course collision graph (Tab.1). Edges are only implicitly represented as pairs of courses that need to be computed at runtime. Hence, our view that mapping abstractions from the two worlds needs full computational power, which is usually not provided by declarative mapping constructs. One of the contributions of this paper is the fact that we present an approach which allows such flexible mappings.

In addition, our on-demand remodularization is object-based rather than class-based. Class-based means that a remodularization which affects a class applies to all instances of a class, whereas object-based means that the remodularization may be created for individual objects on-demand. The advantage of object-based remodularization is twofold. First, we have fine-grained control over the integration process because we can choose for each object whether it should be part of a collaboration or not. Second, the same object (or set of objects) can participate in multiple component instances. For example, a particular course instance can be a vertex in the course collision graph and simultaneously an edge in the "Teacher uses Room" graph (see Tab.1). It can even be a vertex in another course collision graph which is independent from the first one. Hence, object-based remodularization enables us to create multiple independent remodularizations of the same objects. This is indicated in Fig. 3, which outlines the general architecture for using our proposal, by two different remodularizations of the same structure. This was an important requirement on the integration of independent components (recall the different views of the university example in Tab. 1).

An important feature of the model is the loose coupling of the implementation and binding modules which allows to reuse them independently. The implementation of a component type relies on the declarations in the required contract in order to remain oblivious of the potential contexts of use. This renders a component implementation independent of specific applications. By having the expected contract be an integral part of its type, any component implementation carries around a port that makes it pluggable into unknown worlds. Any binding of the collaboration interface can serve as a plug. On the other side, a binding module can also rely on the declarations in the provided contract, remaining oblivous of any potential implementation of the component type being bound. However, by carrying around the provided contract of their type, they can easily be plugged with arbitrary implementations of that type.

To the best knowledge of the authors, our approach is the first one that decouples the component implementations and remodularizations as indicated in Fig. 3 and thus allows us to combine arbitrary implementations with arbitrary bindings of a collaboration interface. It is only after the composition with a binding module that an implementation becomes operative. The gain is that one can write code that is polymorphic with respect to either a component's implementations, or bindings, or both of them, depending on whether the code is written to a certain binding type, a certain implementation type, or to the component type, respectively. In addition, due to an appropriate generalization of common OO concepts (such as types, subtype polymorphism and late binding) from the level of individual classes to the level of sets of collaborating classes, reuse is very naturally supported in both dimensions, component implementation and remodularization.

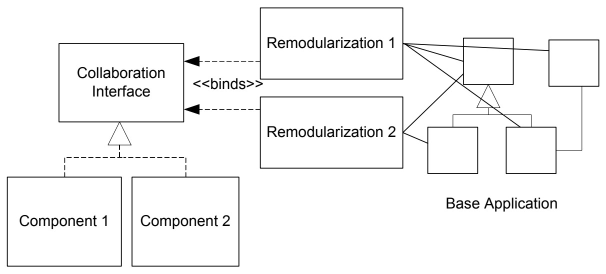

Figure 3: General architecture for collaboration interfaces and on-demand remodularization

The remainder of this paper is organized as follows. Sec. 2 presents shortcomings of current language technology with respect to supporting integration of generic components. Sec. 3 introduces our notions of collaboration interfaces and on-demand remodularization. Sec. 4 outlines future work, in particular our plans towards *fluid aspect-oriented programming*. Sec. 5 discusses related work, and Sec. 6 summarizes the paper.

2. PROBLEM STATEMENT

In this section, we set up the stage for the rest of the paper. The goal is to identify shortcomings of current object-oriented language technology with respect to supporting the development of generic components designed for late integration into various contexts of use. The following sections will present our proposal for coping with these shortcomings. The target of our criticism is the common concept of interfaces as we know it, e.g., from Java. We argue that they lack two important features:

- **Appropriate support for declaring component types as a set of mutually recursive types.** Defining generic components involves in general several related abstractions. We claim that current technology falls short in providing appropriate means to express the different abstractions and their respective features and requirements that are involved in a particular collaboration.
- **Support for bidirectional communication:** Interfaces provide clients with a contract as what to expect from a server object that implements the interface. We say, they express the *provided* contract. In order to define generic components which are decoupled from their potential contexts of use, expressing expectations that a server might have on potential contexts of use is as important. We use the term *expected* contract to denote these expectations. What is needed is support for a loose coupling of client and server, that is (a)

```
interface TreeModel {
  Object getRoot();
  Object[] getChildren(Object node);
  String getStringValue(Object node, boolean selected,
    boolean expanded, boolean leaf, int row, boolean focus);
  }
}

interface TreeGUIControl {
   display();
}

class SimpleTreeDisplay implements TreeGUIControl {
   TreeModel tm;
   display() {
      Object root = tm.getRoot();
      ... tm.getChildren(root) ...
      ...
      // prepare parameters for getStringValue
      ... tm.getStringtValue(...);
      ...
   }
}
```

Figure 4: Simplified version of the Java Swing `TreeModel` interface

decoupling them to facilitate reuse, while (b) enabling them to tightly communicate with each other as part of a whole.

To illustrate these shortcomings, let us have a critical look at a simple example. Fig. 4 shows a simplified version of the `TreeModel` interface in Swing[3], Java's GUI framework [10]. This interface provides a generic description of the data model for a `JTree`, or other GUI tree controls. For illustration purposes, Fig. 4 also presents a pseudo interface for tree GUI controls in `TreeGUIControl`, as well as a pseudo implementation of this interface in `SimpleTreeDisplay` (the latter roughly corresponds to `JTree`).

In our terminology the code in Fig. 4 defines a generic component for displaying arbitrary data structures that can be viewed as trees in a GUI. When this component is used in a particular context, e.g., for object structures that represent arithmetic expressions, it provides to this context the `display` functionality. In turn, it expects from the context a concrete implementation of `getChildren` and `getStringValue`. These operations can only be implemented specifically for a concrete data type to be presented as a tree. That is, `TreeGUIControl` corresponds roughly to what we called the provided contract in the type of our component, while `TreeModel` correspond roughly to what we called the expected contract. The class `SimpleTreeDisplay` represents a sample implementation of the provided contract.

The design in Fig. 4 does actually a good job in decoupling these two contracts. Different implementation of GUI controls can be written to the `TreeModel` interface and can therefore be reused with a variety of concrete implementations of it, i.e., with a variety of data structures. The other way around, any data structure to be displayed is decoupled from a specific tree GUI control (e.g., `JTree`), such that the data structure can be displayed with different GUI tree controls.

[3]Swing separates our interface into two interfaces, `TreeModel` and `TreeCellRenderer`. However, this is irrelevant for the reasoning in this paper.

```
class Expression {
  Expression[] subExpressions;
  String description() { ... }
  Expression[] getSubExpressions() { ... }
}
class Plus extends Expression { ... }
```

Figure 5: Expression Trees

```
class ExpressionDisplay implements TreeModel {
  ExpressionDisplay(Expression r) { root = r; }
  Expression root;
  Object getRoot() { return root; }
  Object[] getChildren(Object node) {
      return ((Expression) node).getSubExpressions();
  }
  String getStringValue(Object node, boolean selected,
    boolean expanded, boolean leaf, int row, boolean focus){
    String s = ((Expression) node).description();
    if (focus) s ="<"+s+">";
      return s;

  }
}
```

Figure 6: Using `TreeModel` to display expressions

So, what is wrong with the approach to specifying generic components exemplified by the design in Fig. 4? The first "bad smell" is the frequent occurrence of the type `Object`. We know that a tree abstraction is defined in terms of smaller tree node abstractions. However, this collaboration of the tree and tree node abstrations is not made explicit in the interface. Since the interface does not state anything about the definition of tree nodes, it has to use the type `Object` for nodes.

The disadvantages of using the most general type, `Object`, are twofold. First, it is conceptually questionable. If every abstraction that is involved in the component definition is only known as `Object`, no messages, beside those defined in `Object`, can be directly called on those abstractions. Instead, a respective top-level interface method has to be defined, whose first parameter is the receiver in question. For example, the methods `getChildren` and `getStringValue` conceptually belong to the interface of a tree node, rather than of a tree. Since the tree definition above does not include the declaration of a tree node, they are defined as top-level methods of the tree abstraction whose first argument is `Object node`.

Second, we loose type safety. Let us have a look at Fig. 5 and Fig. 6. Fig. 5 shows a simple base application for expressions, and Fig. 6 demonstrates how the expression classes can be adapted ('remodularized') to fit in the conceptual world of a `TreeModel`. In our terminology, `ExpressionDisplay` in Fig. 6 represents an implementation of the *expected* contract. Since we use `Object` all the time, we cannot rely on the type checker to prove our code statically safe because type-casts are ubiquitous.

The question naturally raises here: Why didn't the Swing designers define an explicit interface for tree nodes as in Fig. 7 from the very beginning? Well, there are good reasons for this. With the explicit type `NodeTree` it becomes more

```
interface TreeDisplay {
  TreeNode getRoot();
}
interface TreeNode {
  TreeNode[] getChildren();
  String getStringValue(boolean selected,
   boolean expanded, boolean leaf, int row, boolean focus);
}
```

Figure 7: `TreeDisplay` interface with explicitly reified `TreeNode` interface

```
class ExprAsTreeNode
implements TreeNode {
  Expression expr;
  void getStringValue(...) {
    // as before
  }
  TreeNode[] getChildren() {
    Expressions[] subExpr = expr.getSubExpressions();
    TreeNode[] children =
          new TreeNode[subExpr.length];
    for (i = 0; i<subExpr.length; i++) {
       children[i] = new ExprAsTreeNode(subExpr[i]));
    }
    return children;
  }
}
```

Figure 8: Mapping `TreeNode` to `Expression`

difficult to decouple the two contracts, i.e., the data structures to be displayed from the display algorithm. The idea is that the wrapper classes around e.g., `Expression` would look like in Fig. 8. The problem with such kind of wrappers, as also indicated in [8], is that we create new wrappers every time we need a wrapper for an expression. This leads to the *identity hell*: we loose the state and identity of previously created wrappers for the same node. The questionable alternative would be to use hash tables which is not only laborious but does also involve the definition and use of additional classes for maintaining these hashtables, thereby rendering the code more complex and less readable[4].

So far, we discussed problems resulting from the lack of appropriate support for defining multiple related abstractions in one module. Let us now illustrate the problems resulting from the second shortcoming of standard interfaces: missing support for bidirectional communication. Consider for this purpose the `getStringValue()` method in Fig. 4 and Fig. 6. This method has noticeable many parameters that might be of interest when computing a string representation of the node. *Might be.* The sample implementation in Fig. 6 uses only the `selected` parameter and ignores the rest. That means, the tree GUI control, which calls this method on the `TreeModel` interface, has to perform expensive computations to obtain the parameter values for this method (see implementation of `SimpleTreeDisplay::display()` in Fig. 4), although they might be rarely all used.

This is a typical case where we would like to establish a bidirectional communication between the two contracts of the tree displaying component. Here we would like `ExpressionDisplay.getStringValue` to explicitly ask the tree GUI control to compute only relevant values for it, like `selected` or `hasFocus`, implying the GUI control interface provides respective operations. Recall that the GUI control interface corresponds to the provided interface of our generic component for displaying arbitrary data structures that can be viewed as trees in a GUI. As for now, the interfaces are completely separated into (`TreeModel` and `TreeGUIControl`), and there is nothing in the design that would suggest their tight relation as two faces of the same abstraction. As such, there is no build-in support for bidirectional communication between their respective implementations. Build-in means by the virtue of implementing two faces of the same abstraction, which serves as the implicit communication channel.

One can certainly achieve the desired communication by additional infrastructure (e.g., via cross-references) which has to be communicated to the respective programmers. However, we think that bidirectional communication is such a natural and frequent concept that the overhead that is necessary to enable bidirectional communication with conventional interfaces is too high. Please note that the additional `TreeNode` interface would also be of no help concerning the bidirectional communication problem exemplified by the `getStringValue()` method.

The third point is the fact that it is difficult and awkward to associate state with abstractions like our tree nodes. We might want to associate state with tree nodes in both the `ExpressionDisplay` class in Fig. 6 and also inside the tree gui control. For example, we might want to cache the computed string value or children in Fig. 6, because the recomputation might be expensive. In the gui control itself, we might want to associate state like whether a tree node is selected or not or its position on the screen with the respective tree node. The only means to associate state with tree nodes is to make extensive use of hash tables, which is laborious and awkward.

3. CORE CONCEPTS

In this section, we will give an overview of the concepts that comprise our model by means of the `TreeDisplay` example from the previous section.

3.1 Collaboration Interfaces, their Implementations and Bindings

In order to cope with the problems discussed in Sec. 2 we propose the notion of *collaboration interfaces* (*CI* for short), which differ from standard interfaces in two ways. First, CIs introduce the `provided` and `required` modifiers to annotate operations belonging to the provided and the expected contracts, respectively, hence supporting bidirectional interaction between clients and servers. Second, CIs exploit interface nesting in order to express the interplay between multiple abstractions participating in the definition of a generic component.

For illustration, the CI `TreeDisplay` that bundles the definition of the generic tree displaying functionality from Sec. 2 is shown in Fig. 9. As an example for the "reification" of provided and expected contract, consider the methods `TreeDisplay.display()` and `TreeDisplay.getRoot()` in Fig. 9. Any tree display object is able to display itself on the request of a client - hence the `provided` modifier for `TreeDisplay.display`. However, in order to do so, it ex-

[4]In fact, Swing offers a `TreeNode` interface similar to the one in Fig. 7. However, classes that define data structures to be displayed as tree nodes should anticipate this and explicitly implement the interface.

```
interface TreeDisplay {
  provided void display();
  expected TreeNode getRoot();

  interface TreeNode {
    expected TreeNode[] getChildren();
    expected String getStringValue();
    provided display();
    provided boolean isSelected(),
    provided boolean isExpanded();
    provided boolean isLeaf();
    provided int row();
    provided boolean hasFocus();
  }
}
```

Figure 9: Collaboration interface for TreeDisplay

pects a client specific way of how to access the root tree node. What the root of a displayable tree will be depends on (a) which modules in a concrete deployment context of `TreeDisplay` will be seen as tree nodes and, (b) which one of them will play the role of the root node. Hence, the declaration of `getRoot` with the expected modifier. `TreeDisplay` comes with its own definition of a tree node: The CI `TreeNode` is nested into the declaration of `TreeDisplay`. Please note that nesting of bidirectional interfaces in our approach has a much deeper semantics than usual nested classes and interfaces in Java: the nested interfaces are namely *virtual types* as in [6]. We will elaborate on that in Sec. 3.4.

The categorisation of the operations into expected and provided comes with a new model of what it means to implement an interface. We explicitly distinguish between *implementing* an interface's provided contract and *binding* the same interface's expected contract. Two different keywords are used for this purpose: `implements`, respectively `binds`. In the following, we refer to classes that are declared with the `implements` keyword as *implementation classes*. Similarly, we refer to classes that are declared with the `binds` keyword as *binding classes*

An implementation class of a CI must (a) implement all `provided` methods of the CI and (b) provide an implementation class for each of the CI's nested interfaces. In doing so, it is free to use respective `expected` methods. In addition, an implementation class may or may not add additional methods and state to the CI's abstractions it implements. Fig. 10 shows a sample tree GUI control that implements `TreeDisplay`. The class `SimpleTreeDisplay` implements the only provided operation of `TreeDisplay`, `display()`, by forwarding to the result of calling the expected operation `getRoot()`. In addition to implementing `display()`, `SimpleTreeDisplay` must also provide a nested class that implements `TreeNode`. The correspondence between a nested implementation class and its corresponding nested interface is based on name identity – `SimpleTreeDisplay` e.g., defines a class named `TreeNode` which is the implementation of the nested interface with the same name in `TreeDisplay`. This nested class has to implement all `provided` methods of the `TreeNode` interface, e.g., `display()`. The declaration of the instance variable `boolean selected` and the corresponding query operation `isSelected` in `SimpleDisplay.TreeNode` are examples of new declarations added by an implementation class. Just as nested interfaces, all nested implementation classes are virtual types (see Sec. 3.4).

```
class SimpleTreeDisplay implements TreeDisplay {
  void onSelectionChange(TreeNode n, boolean selected) {
    n.setSelected(true);
  }
  void display() { getRoot().display(); }
  class TreeNode {
    boolean selected;
    ...
    boolean isSelected() { return selected; }
    // other provided methods similar to selected
    void setSelected(boolean s) { selected =s;}
    void display() {
      ... TreeNode c = getChildren()[i];
      ... paint(position, c.getStringValue());
      ...
    }
  }
}
```

Figure 10: A sample implementation of `TreeDisplay`

A binding class of a CI must (a) implement all `expected` methods of the CI, and (b) provide zero or more binding classes for each of the CI's nested interfaces (we may have multiple bindings of the same interface, see subsequent discussion). Just as implementation classes can use their respective expected facets, the implementation of the expected methods of a CI and its nested interfaces can also call methods declared in the respective provided facets. The process of binding a CI instantiates its nested types for a concrete usage scenario of the generic functionality defined by the CI. Hence, it is natural that in addition to their provided facets, binding classes also use the interface of abstractions from that concrete usage scenario. We say that bindings wrap abstractions from the world of the concrete usage scenario and map them to abstractions from the generic component world.

For illustration, the class `ExpressionDisplay` in Fig. 11 shows an example of binding the generic `TreeDisplay` CI from Fig. 9 for the concrete usage scenario, in which `Expression` structures are to be viewed as the trees to display. First, `ExpressionDisplay` binds the nested type `TreeNode` as shown in the nested class `ExprTreeNode`. The latter implements all expected methods of `TreeNode` by using (a) the provided facet of `TreeNode`, and (b) the interface of the class `Expression` (via the instance variable `e`). Consider e.g., the implementation of the method `ExprTreeNode.getStringValue()`, which calls both `TreeNode.hasFocus()` as well as `Expression.getDescription()`.

In addition to binding `TreeNode`, `ExpressionDisplay` also implements the method `getRoot()` - the only method declared in the *expected* facet of `TreeDisplay`. Here is where the reference `root` to the `Expression` object to be seen as the root of the expression structure to display is transformed into a `TreeNode` by being wrapped into an `ExprTreeNode` object. Please note that this wrapping does not happen via an ordinary constructor call - `new ExprTreeNode(root)` in this case -, but rather by means of the *wrapper recycling* call `ExprTreeNode(root)`. We will elaborate on the concept of *wrapper recycling* in a moment.

The careful reader should have noticed that we do not use identical names for establishing the correspondence between a binding class and its corresponding nested interface, as

```
class ExpressionDisplay binds TreeDisplay {
  Expression root;

  public ExpressionDisplay(Expression rootExpr) {
     root = rootExpr;
  }

  TreeNode getRoot() {
     return ExprTreeNode(root);
  }

  class ExprTreeNode binds TreeNode {
    Expression e;
    ExprTreeNode(Expression e) { this.e=e;}
    TreeNode[] getChildren() {
      return ExprTreeNode[](e.getSubExpressions());
    }
    String getStringValue() {
      String s = e.description();
      if (hasFocus()) s ="<"+s+">";
      return s;
  }
}
```

Figure 11: Binding of `TreeDisplay` for expressions

we did with implementing classes (`ExprTreeNode` in Fig. 11 binds `TreeNode`, but is itself not called `TreeNode`). As indicated in Sec. 1, different bindings of the same interface might be needed, if we have different abstractions in the concrete usage scenario that have to be mapped to the same component abstraction. This was illustrated by the "Student knows Teacher" graph from Tab. 1: Both, students and teachers, play the role of vertices in this graph. To cope with such multiple bindings of the same interface, the identification by name that we had with implementation classes is not carried over to binding classes. This enables multiple different bindings of the same component type to different base types without running into conflicts, since the bindings can still be discriminated by their names.

In Sec. 1 we gave an example that illustrated why declarative mapping constructs as in [27, 20, 21] are not sufficient to express arbitrary on-demand remodularizations. In general, the full computational power of an object-oriented language is needed for this purpose. For this reason, our approach to specifying remodularizations is rather "manual". In fact, binding classes and their nested classes are "almost standard" classes. "Almost" stands for two differences. First, nested binding classes are also virtual types (see Sec. 3.4). Second, they make use of the notion of *wrapper recycling*, which we discuss next.

3.2 Wrapper Recycling

Wrapper recycling is our mechanism to escape the wrapper identity hell mentioned in Sec. 2. It is a concept on how to create and maintain wrapper instances, and a way to navigate between abstractions of the component world and abstractions of the base world - the concrete usage scenario world -, ensuring that the same (identical) wrapper instance will always be retrieved for a set of constructor arguments. This way the state and the identity of the wrappers is preserved.

Syntactically, wrapper recycling refers to the fact that, instead of creating an instance of a wrapper `W` with a standard `new W(constructorargs)` constructor call, a wrapper is retrieved with the construct `outerClassInstance.W(constructorargs)`. For illustration consider once again the expression `return ExprTreeNode(root)` in the method `ExpressionDisplay.getRoot()` in Fig. 11. We already mentioned in the previous section that the expression in the return statement is not a standard constructor call, but rather a wrapper recycling operator. We use the usual Java scoping rules, i.e., `return ExprTreeNode(root)` is just an abbreviation for `return this.ExprTreeNode(root)`.

The idea is that we want to avoid creating a new `ExprTreeNode` wrapper each time the method `getRoot()` is called on an `ExpressionDisplay`. The call to the wrapper recycling operation `ExprTreeNode(root)` is equivalent to the corresponding constructor call, only if a wrapper for `root` does not already exist, ensuring that there is a unique `ExprTreeNode` wrapper for each expression within the context of the enclosing `ExpressionDisplay` instance. That is, two subsequent wrapper retrievals for an expression `e` yield the same wrapper instance - the identity and state of the wrapper are preserved.

This is due to the semantics of a wrapper recycling call, which is as follows: The outer class instance maintains a map `mapW` for each nested wrapper class `W`. An expression `outerClassInstance.W(wrapperargs)` corresponds to the following sequence of actions:

1. Create a compound key for the constructor arguments, lookup this key in `mapW`.

2. If the lookup for the key fails, create an instance of `outerClassInstance.W` with the annotated constructor arguments, store it in the hash table `mapW`, and return the new instance. Otherwise return the object already stored in `mapW` for the key.

The wrapper recycling call `ExprTreeNode[](...)` in the method `Expr.TreeNode.getChildren()` in Fig. 11 is an example for the syntactic sugar we use to express wrapper recycling of arrays, namely an automatic retrieval of an array of wrappers for an array of base objects.

A naive implementation of wrapper recycling in a language with garbage collection would imply a memory hole because wrapped objects would never be collected by the garbage collector. However, this can easily be reconciled by more advanced memory management techniques such as weak references and reference queues. Java, for example, has a standard API class `WeakHashMap` that could be used instead of a usual map.

3.3 Composing Bindings and Implementations

Both classes defined in Fig. 10 and 11 are not operational, i.e., cannot be instantiated, even if they are not annotated as abstract. These classes are indeed not abstract, since they are complete implementations of their respective contracts. The point is that the respective contracts are parts of a whole and make sense only within a whole. Operational classes that completely implement an interface are created by composing an implementation and a binding class, using the composition operator `+`. This is illustrated by the class `SimpleExpressionDisplay` in Fig. 12. Only such compound classes are allowed to be instantiated by the compiler. For instance, Fig. 12 also shows sample code that instantiates and uses the compound class `SimpleExpressionDisplay`.

```
class SimpleExpressionDisplay =
          SimpleTreeDisplay + ExpressionDisplay;
...

Expression test = new Plus(new Times(5, 3), 9);
TreeDisplay t = new SimpleExpressionDisplay(test);
t.display();
```

Figure 12: Creating and using compound classes

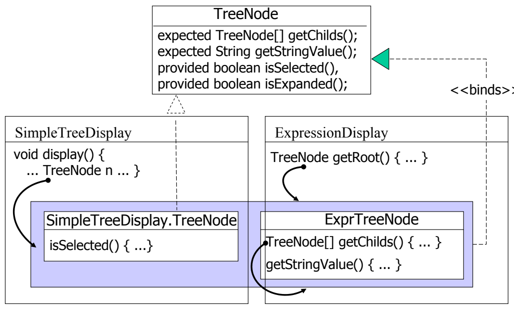
TreeNode
expected TreeNode[] getChilds();
expected String getStringValue();
provided boolean isSelected(),
provided boolean isExpanded();
<<binds>>
SimpleTreeDisplay
void display() {
... TreeNode n ... }
ExpressionDisplay
TreeNode getRoot() { ... }
SimpleTreeDisplay.TreeNode
isSelected() { ...}
ExprTreeNode
TreeNode[] getChilds() { ... }
getStringValue() { ... }

Figure 13: Type rebinding in compound classes

Combining two classes with "+" means that we create a new compound class within which the respective implementations of the `expected` and `provided` methods are combined. The same combination also takes place recursively for the nested classes: All nested classes with a `binds` declaration are combined with the corresponding implementation from the component class. The separation of the two contracts, their independent implementation, and the dedicated late composition, allows us to freely reuse implementations of the two contracts in arbitrary compositions. We could combine `SimpleTreeDisplay` with any other binding of `TreeDisplay`. Similarly, `ExpressionDisplay` could be combined with any other implementation of `TreeDisplay`. Sec. 4 will have more to say about the reuse dimensions and the flexibility supported by our approach.

Note that the overall definition of the nested type, e.g., `TreeNode`, depends on the concrete composition of implementation and binding types within which the type is used. This does not only affect the external clients, but also the internal references. For instance, references to `TreeNode` within `ExpressionDisplay` and `SimpleDisplay` are rebound to the composed definition of `TreeNode` in `SimpleExpressionDisplay`, as illustrated in Fig. 13. Their meaning would be different in another compound class, e.g., resulting from composing `SimpleDisplay` with another binding class, or `ExpressionDisplay` with another implementation class. This is a natural consenquence of the fact that nested types introduced by the collaboration interfaces are virtual types, on which we will elaborate in the following.

3.4 Virtual Types

In our approach, all types that are declared as nested interfaces of a CI and all classes that implement or bind such interfaces (including classes that extend the latter) are *virtual types* and *virtual classes*, respectively [15]. In the context of this paper, we use the notion of virtual types of the family polymorphism approach [6]. This means: (a) similar to fields and methods, types also become properties of objects of the class in which they are defined, and consequently (b) their denotation can only be determined in the context of an instance of the enclosing class. Hence, the meaning of a virtual type is late bound depending on the receiver object that executes when the virtual type at hand is referenced.

Consequently, all type declarations, constructor calls, and wrapper recycling calls for virtual types/classes within a CI are actually always annotated with an instance of the enclosing class. That is, type declarations and constructor invocations are always of the form `enclInst.MyVirtual x`, respectively `enclInst.MyConstructor()`. Similarly, wrapper recycling calls are also always of the form `outerClassInstance.W(constructorargs)` and not simply `W(constructorargs)`. For the sake of simplification, we apply the scoping rules common for Java nested classes also to type declarations and constructor or wrapper recycling calls: A call `OuterClass.this.W(constructorargs)` can be shortened to `W(constructorargs)`, and the type declaration `OuterClass.this.W` can be shorted to `W` as long as there are no ambiguities. This scoping rule applies to all type declarations and wrapper recycling calls that have appeared so far in this paper.

For instance, all references to `ExprTreeNode` in Fig. 11 should be read as `ExpressionDisplay.this.ExprTreeNode`. The implication is that the meaning of any reference to the type name `ExprTreeNode` within the code of `ExpressionDisplay` will be bound to the compound class that combines `ExpressionDisplay.ExprTreeNode` with the implementation class of `TreeNode` that is appropriate in the respective execution context. For example, in the context of a `SimpleExpressionDisplay` as in Fig. 12, `ExprTreeNode` will be bound to the compound class `ExpressionDisplay.ExprTreeNode + SimpleTreeDislay.TreeNode`. The same references will be bound differently if they occur in the execution context of an object of some subclass of `ExpressionDisplay` or in the context of a different implementation class. The same also applies to nested implementation and compound classes.

The rationale behind using virtual types lies in their power with respect to supporting reuse and polymorphism, as argued in [6]. The advantages with respect to the degree of reuse we gain in our context will be discussed in Sec. 4. At this point, we will rather shortly discuss how our specific use of virtual types (borrowed from [6]) does *not* suffer from covariance problems usually associated with virtual types, as for example the virtual type proposal in [28], which requires runtime type checks. If we have a virtual type in a contravariant position, as for example the argument type of `setRoot` in Fig. 14, type safety is still preserved, because subsumption is disallowed if the enclosing instances are not identical. In order to make the approach sound, all variables that are used as part of type declarations have to be declared as `final` because otherwise the meaning of a type declaration might change due to a field update. For illustration consider the declaration of the variable `ed` in the sample code in Fig. 14. It is used as part of a type declaration for the variable `t` and is therefore declared as `final`. For more details on typing issues we refer to [6].

3.5 Object Constructors

Having classes with "splitted" code, as in our division into a binding part and an implementation part, the question of

```
Expression e = ...;
final ExpressionDisplay ed =
                   new SimpleExpressionDisplay(e);

...

// let FileSystemDisplay be a binding of
// TreeDisplay to the file system structure
SimpleFileSystemDisplay =
             SimpleTreeDisplay + FileSystemDisplay;

FileSystem fs = ... ;
final FileSystemDisplay fsd =
       new SimpleFileSystemDisplay(fs);

...

ed.TreeNode t = ed.getRoot();

fsd.setRoot(t); // Type error detected by typechecker!
        // sd.TreeNode is not subtype of ed.TreeNode
```

Figure 14: Type safety due to family polymorphism

object construction and constructor calls arises. Which site (binding or implementation) should be able to implement, respectively call, constructors? An important prerequisite in the following discussion is that we assume that – in general – constructors have arguments.

Allowing both sites to implement and call constructors would be unsound because every site would only call its own constructors (it does not know about the existence of the other-side constructors), and therefore invariants that are established in the constructors of the other site will not hold since the constructor of the other site will never be called - implicit constructor invocation is not possible if the constructor requires arguments.

Our point of view is that only the binding site should implement constructors because the binding site needs to establish links to base objects (which are adapted to the role they play in the particular collaboration of a generic component) in order to fulfill its purpose. If the implementation site needs to create objects, this can easily be done by specifying corresponding `expected` factory methods in the collaboration interface which can be called from the implementation site and are implemented at the binding site. Therefore, only the binding classes can implement constructors, and these constructors are also the constructors that are available in the compound classes that combine an implementation class with a binding class by means of the `+` operator.

Although this point (object creation) seems to be only marginal, it has an important conceptual implication that is related to *symmetry*. At first, collaboration interface implementation and binding seem to be rather symmetric, but object creation creates an important asymmetry. A consequence of this asymmetry is that we can have only one implementation of a bidirectional interface, but we may have multiple different bindings of a bidirectional interface; we can select among these bindings by means of different constructors.

3.6 Interim Evaluation of the Model

As an interim result, let us compare the way the generic tree display functionality and its instantiation for expressions was modeled with our model to the conventional solution discussed in Sec. 2.

- Other than the Swing interface in Fig. 4, we do not need to use `Object`; every item is well-typed and we do not need type casts. The methods that are conceptually part of the interface of tree nodes, are expressed as methods of a dedicated nested interface.
- Due to bidirectional interfaces, we do not have the problem related to the `getStringValue()` parameters: The implementation of this method, as in Fig. 11, causes the computation of only those values about the state of displaying that are really needed by means of calling appropriate methods in the `provided` interface.
- It is easy to associate additional state with tree nodes. For example, the `TreeNode` implementation in Fig. 10 adds a `selected` field, and the `TreeNode` binding in Fig. 11 could as well have added extra state to `ExprTreeNode`.

4. DIMENSIONS OF REUSE

In this section we want to elaborate on the degrees of reuse and polymorphism supported by our proposal. For this purpose, we will use the graph example from Sec. 1, since it is better suited to demonstrate the advantages of our approach.

4.1 Component Type Hierarchies

The first kind of reuse supported by our model is along the dimension of component types. The key object-oriented notion of subtyping between individual interfaces extends very naturally to our nested collaboration interfaces. New CIs can be defined as extensions of already existing CIs via the `extends` clause. The new CI inherits all nested type definitions and provided/expected methods of its parent CI. The inheriting CI can than add new nested type definitions and expected/provided method declarations. In addition, the inherited nested types can be refined by defining interfaces with the same name annotated by the modifier `override`. An "overriding" nested type inherits all declarations of the nested type being overridden and can add new declarations.

For illustration, Fig. 15 shows three sample collaboration interfaces for graphs. The top interface `Graph` defines the general graph abstractions and properties of these abstractions. The other two interfaces, `ColoredGraph` and `MatchedGraph`, refine `Graph` by adding methods or refining inherited nested interfaces of `Graph`[5].

The refinement of nested types has pretty much the semantics of standard inheritance on types. So, why the new syntax - the keyword `override` rather than the familiar `extends`? The reason becomes clear once you recall that our nested types are virtual types, rather than standard types as e.g., Java interfaces. Other than an ordinary `extends` declaration, an `override` declaration does not create a *new* type with a *n*ew name but overrides the definition of the inherited type. The typing implications of these virtual types were explained in the previous section.

[5]The purpose of the `ColoredGraph.Edge.getBadness()` method is to have a measure of how troublesome a particular edge is with respect to minimum coloring, meaning that if an edge with a high badness would be removed, it is likely that we can color the graph with less colors.

```
interface Graph {
  interface Vertex {
    expected Edge[] getEdges();
  }
  interface Edge {
    expected Vertex getV1();
    expected Vertex getV2();
  }
}

interface ColoredGraph extends Graph {
  provided computeMinimumColoring(Vertex v[]);
  override interface Vertex {
    expected void setColor(int c);
    expected int getColor();
  }
  override interface Edge {
    provided float getBadness();
}

interface MatchedGraph extends Graph {
  provided computeMaximumMatching(Vertex v[]);
  override interface Edge {
    expected void setMatched(boolean b);
    expected boolean isMatched();
  }
}
```

Figure 15: Graph collaboration interfaces

4.2 Implementation Hierarchies

Fig. 16 shows two implementation classes for the `ColoredGraph` interface from Fig. 15, `SuccessiveAugmentationColoring` and `SimmulatedAnnealingColoring`, each employing a different algorithm for graph coloring. By being implementation classes, each of them provides implementations for the `provided` methods of `ColoredGraph`, using the declared `expected` methods. In addition, they may add new declarations. For example, `SuccessiveAugmentationColoring` adds a field `temp_color` to `Vertex`. The association of the nested classes with the corresponding nested interface in `ColoredGraph` happens by common names, as already explained in Sec. 3.1

Similar to interface refinement as in Fig. 15, it is also possible to refine implementation classes, whereby the definitions of the nested classes can again be refined with the `override` modifier. In other words, the subclassing and subtyping relations between individual classes in standard OO are naturally carried over to the implementation classes of CIs; again with the important difference that our nested classes are virtual types as explained in the previous section. For example, the commonalities between the two different coloring algorithms could be factored out into a common superclass as in Fig. 17.

4.3 Binding Hierarchies

In the following we elaborate on some advanced issues related to CI bindings that could not be appropriately demostrated by the simple example of the previous section. Furthermore, we discuss the extent to which reuse is enabled along the dimension of binding classes.

Recall our claim that binding a CI to a concrete application implies an *on-demand remodularization* of the application for which simple declarative mappings are insufficient. With the more sophisticated graph example, we are now able

```
class SuccessiveAugmentationColoring
implements ColoredGraph {
  // successive augmentation coloring algorithm
  void computeMinimumColoring(Vertex v[]) {
    // successive augmentation coloring algorithm
    ... Edge e[] = v[i].getEdges(); ...
    ... Vertex w = e[j].getV2();
    ... if (w.isLegalColor(color)) w.temp_color = color;
    ... e[n].setBadness(badness); ...
    // commit final coloring
    for (int k=0; k<v.length;k++)
      v[k].setColor(v[k].temp_color);
  }
  class Vertex {
    int temp_color;
    boolean isLegalColor(int color) {
      Vertex neighbor[] = ...;
      for (int i=0;i<neighbor.length;i++)
        if (neighbor[i].getColor() == color) return false;
      return true;
    }
  }
  class Edge {
    float badness;
    float getBadness() { return badness; }
    void setBadness(float b) { badness = b; }
  }
}


class SimulatedAnnealingColoring
implements ColoredGraph {
  // Simulated Annealing coloring algorithm
  void computeMinimumColoring(Vertex v[]) {
    ...
  }
  ...
}
```

Figure 16: Different Coloring Algorithms

to illustrate, how our proposal copes with this requirement. For this purpose, Fig. 18 shows a binding class that transforms the scheduling graph structure hidden within the university class structure to the class structure that is required by our graph algorithms. The nested classes `CourseCollision` and `CourseVertex` are remodularization wrappers around base objects. The class `CourseCollision` implements the expected interface of `ColoredGraph.Vertex` by wrapping an object `c` of type `Course`, while `CourseVertex` implements the expected interface of `ColoredGraph.Edge` by wrapping two objects of type `Course`, `c1` and `c2`.

Please note that the class `CourseCollision` wraps *two* courses because there is no dedicated abstraction for course collisions in the base application. This scenario illustrates one part of our claim that simple declarative role mappings are not sufficient. Binding a CI type is, in general, not a simple equation of it with a type in the base application. It rather might imply the collaboration of several instances of the same or of different base types.

The example at hand also allows us to illustrate how our proposal deals with the requirement for multiple bindings of the same interface to different abstractions in the base application - the other part of our claim that simple declarative role mappings are not sufficient. This was exemplified by the "Student knows Teacher" graph in Tab. 1, where both, students and teachers, play the role of vertices in this graph.

```
abstract class AbstractColoring
implements ColoredGraph {
  class Vertex { ... }
  class Edge { float badness; ... }
}

class SuccessiveAugmentationColoring
extends AbstractColoring {

  // Successive augmentation coloring algorithm
  void computeMinimumColoring(Vertex v[]) { ... }
  override class Vertex { ... }
}

class SimulatedAnnealingColoring
extends AbstractColoring {

  // Simulated Annealing coloring algorithm
  void computeMinimumColoring(Vertex v[]) { ... }
  override class Vertex { ... }
}
```

Figure 17: Factoring out the commonalities between the coloring algorithms

```
class SchedulingGraph binds ColoredGraph {
  class CourseVertex binds Vertex {
    Course c;
    Edge[] cachedEdges;
    CourseVertex(Course c) { this.c = c; }
    Edge[] getEdges() {
      if (cachedEdges == null) {
        Vector tc = c.getTeacher().getCourses();
        tc.append(c.getStudentYear().getCourses());
        cachedEdges = new Edge[tc.length];
        for (int i=0;i<tc.length;i++) {
          Course x = (Course) tc[i];
          cachedEdges[i] = CourseCollision(c,x);
        }
      }
      return cachedEdges;
    }
    void setColor(int color) {
      c.timeSlot = TimeSlots.getSlot(color);
    }
  }
  class CourseCollision binds Edge {
    Course c1,c2;
    CourseCollision(Course c1, Course c2) {
      this.c1=c1; this.c2 = c2;
    }
    Vertex getV1() { return CourseVertex(c1); }
    Vertex getV2() { return CourseVertex(c2); }
  }
}
```

Figure 18: Binding for scheduling graph

```
class StudentKnowsTeacherGraph binds Graph {

  class StudVertex binds Vertex {
    final Student s;
    StudVertex(Student s) {this.s=s;}
    ...
  }

  class TeacherVertex binds Vertex {
    Teacher t;
    TeacherVertex(Teacher t) { this.t = t; }
    ...
  }

  ... Vertex v1 = StudVertex(aStudent); ...
  ... Vertex v2 = TeacherVertex(aTeacher); ...
}
```

Figure 19: Multiple bindings of the same interface

In the previous section, we indicated that in our model, this can be expressed by implementing different bindings of the same interface with different names, which is now illustrated in Fig. 19. Here we have multiple bindings of the interface `Vertex` without, however, introducing ambiguity, because the bindings can still be discriminated by the different names, `StudVertex` and `TeacherVertex`, respectively.

Finally, we would like to use the more sophisticated example to discuss more advanced issues of wrapper recycling. Consider the wrapper recycling calls `CourseCollision(c,x)` in `CourseVertex.getEdges` in Fig. 18, which ensure that there is only one unique instance of `CourseCollision` for each pair `(c1,c2)` of courses. In this example, an undirected edge in the course collision graph is represented by two directed edges, therefore a wrapper recycling call `CourseCollision(c1,c2)` will in general yield a different wrapper than `CourseCollision(c2,c1)` – In other words: wrapper recycling takes the order of the constructor arguments into account. If we want to disregard the order of the arguments, this can be done with an appropriate data structure. For example, a direct representation of undirected edges would also be possible if we would pass a *set* with the two courses as elements in the `CourseCollision` constructor calls instead of the ordered pair of courses.

Now that we have illustrated advanced issues of bindings, let us focus on the reuse supported by our proposal along this dimension. For this purpose, Fig. 20 shows a completely different view of the university application as a graph. The classes defined in Fig. 20 remodularize the university application to present the student contacts graph. In this graph, students play the role of vertices and two vertices are connected if the students visit a joint course. The class `StudContactsGraph` represents the general remodularization to the `Graph` collaboration, while `StudContactsColoredGraph` and `StudContactsMatchedGraph` refine this class in order to specialize the collaboration to `ColoredGraph` and `MatchedGraph`[6], respectively. A minimum coloring in the student contacts graph would represent maximum groups of students that do not know each other and would therefore be good candidates for joint exams with little cheating opportunities. A maximum matching, on the other hand, would be helpful to assign the students to two person apartments, such that

[6]The code for the `MatchedGraph` CI and its implementation are not shown but are analogous to the coloring example

```
class StudContactsGraph binds Graph {
  class StudVertex binds Vertex {
    final Student s;
    StudVertex(Student s) {this.s=s;}
    Edge[] getEdges() {
      ... Student t = ... ;
      ... e[i] = StudContact(s,t);
      return e;
    }
  }
  class StudContact binds Edge {
    Student s,t;
    StudContact(Student s, Student t) {
      this.s = s; this.t = t;
    }
  }
}
class StudContactsColoredGraph extends StudContactsGraph
                              binds ColoredGraph {
  override StudVertex {
    void setColor(int c) {
      Exam.joinGroup(s,c);
    }
  }
}
class StudContactsMatchedGraph extends StudContactsGraph
                              binds MatchedGraph {
  override StudContact {
    void setMatched(boolean b) {
      Rooms.getFreeApartment().assignStudents(s,t);
    }
  }
}
```

Figure 20: Alternative bindings of `ColoredGraph` and `MatchedGraph`

most students are pooled together with a person they know.

The sample code in Fig. 20 is presented for illustrating two nice features of our model. First, together with the code in Fig. 18, it demonstrates how two “worlds” of types can be multiply mapped to each other without ever being changed. The second feature is again due to the seamless integration of our new concepts into the standard object-oriented concepts of classes, inheritance and subtype polymorphism. Inheritance allows us to reuse `StudContactsGraph` in the definition of both `StudContactsColoredGraph` and `StudContactsMatchedGraph`. Similar to the refinements of nested interfaces in Fig. 15, and the refinements of nested implementation classes in Fig. 17, the nested bindings `StudVertex` and `StudContact` of `StudContactsGraph` can be refined with an `override` declaration, as illustrated by `StudContactsColoredGraph.StudVertex` and `StudContactsMatchedGraph.StudContact`.

4.4 Polymorphism and the + Operator

As introduced in Sec. 3.3, implementations and bindings of a CI can be freely combined by means of the “+” operator. In terms of the graph example, this is illustrated in Fig. 21, where two different complete realizations of `ColoredGraph` (cf. Fig. 15) are defined by combining the same binding class, `SchedulingGraph`, with two different implementation classes, `SuccessiveAugmentationColoring` and `SimulatedAnnealingColoring`.

Both combinations, `SucAugSched` and `SimAnSched`, are subtypes of their common binding part, `SchedulingGraph`,

```
class SucAugSched =
  SuccessiveAugmentationColoring + SchedulingGraph;
class SimAnSched =
  SimulatedAnnealingColoring + SchedulingGraph;
...
final SchedulingGraph sg =
  wantSucAug ? new SucAugSched() : new SimAnSched();
sg.computeMinimumColoring( sg.CourseVertex[](courses) );
```

Figure 21: Demo code

```
class SucAugStudContacts =
   SuccessiveAugmentationColoring + StudContactsColoredGraph;
class SimAnStudContacts =
   SimulatedAnnealingColoring + StudContactsColoredGraph;
class Matching1StudContacts =
   MatchingAlgorithm1 + StudContactsMatchedGraph;
class Matching2StudContacts =
   MatchingAlgorithm2 + StudContactsMatchedGraph;
```

Figure 22: Free combination of components and connectors

and can therefore uniformly be used whenever an object of type `SchedulingGraph` is expected. One can view the type `SchedulingGraph` as being parameterized by the provided contract of its CI and the + operator as an application operator that binds an actual value to the formal parameter. `SucAugSched` and `SimAnSched` are two instantiations of `SchedulingGraph` that differ from each other on the coloring algorithm. This allows us to write code like the `computeMinimumColoring()` call in Fig. 21 polymorphically with respect to the coloring algorithm used.

Other examples of this kind, i.e., the same binding classes being combined with different implementation classes can be found in Fig. 22. Just as `SchedulingGraph`, `StudContactsColoredGraph` binding from Fig. 20 can be combined with any of the `ColoredGraph` implementations presented in Fig. 16. Similarly, `StudContactsMatchedGraph` can be combined with an arbitrary matching algorithm that implements `MatchedGraph` (adumbrated in Fig. 22). Both `SucAugStudContacts` and `SimAnStudContacts` are subtypes of `StudContactsColoredGraph` and can be used everywhere it is expected. Similarly, `Matching1StudContacts` and `Matching2StudContacts` are subtypes of `StudContactsMatchedGraph`.

On the reverse side, one could think of coloring algorithms, i.e., of implementation types, as being parameterized with the expected facet of their CI. Hence, any operation written to an implementation type, is naturally polymorphic with respect to all bindings of that implementation type's CI. For instance, `SucAugStudContacts` from Fig. 22 and `SucAugSched` from Fig. 21 represent two instantiations of `SuccessiveAugmentationColoring`, i.e., both are subtypes of the latter.

To summarize the typing relationships: If we have a class `C = A+B` with implementation class `A` and binding class `B` which communicate over a common collaboration interface `CI`, then `C` is a subtype of both `A` and `B`, which are subsequently both subtypes of `CI`.

Since `A` and `B` are independent classes, we have to deal with conflicts which are caused by accidental name clashes of methods in `A` and `B`. We resolve these possible conflicts by hiding all methods of `A` and `B`, which are not already in the collaboration interface `CI`, in the context of a reference of type `C`. For example, if both `A` and `B` would introduce a method `m()`, then `C c = ...; c.m();` would be an illegal call, whereas `A a = c; a.m();` would be legal, thereby eleminating all possible ambiguities and conflicts.

To recap, we can create completely different and independent mappings of a base structure (e.g., university administration) to a particular component structure (e.g., graph) and combine them with yet a range of different implementations of the component (e.g., different coloring algorithms). In general, the role or task that base objects play in a particular collaboration is not static but depends on the context within which the collaboration is used, e.g., a course is a vertex in the course collision graph, and the same course is an edge in the "teacher uses room" graph (see Tab. 1). This is also an advantage of our model over previous more static approaches.

4.5 Section Summary

To summarize the chapter, we want to recall the different dimensions of polymorphism and reuse that are possible in our approach:

- **Collaboration interface dimension**: A hierarchy of collaboration interfaces can be defined, such as the `Graph` interface which is refined by `ColoredGraph` and `MatchedGraph` (see Fig. 15).
- **Component dimension**: Multiple independent implementations of a collaboration interface are possible, such as the different coloring implementations in Fig. 16. Component implementations can reuse other component implementations to implement more specialized collaboration interfaces via inheritance. For example, the communalities of the two coloring algorithms in Fig. 16 can be outsourced into a common superclass (Fig. 17).
- **Connector dimension**: Multiple independent bindings of a collaboration interface to the same or different applications can co-exist, such as the course collision and the student contacts remodularizations in Fig. 18 and 20, respectively. Inheritance among connectors, such as in Fig. 20, allows to reuse existing remodularization specifications when binding more specialized collaboration interfaces.
- **Bound component dimension**: The bound component is a subtype of both the component and the connector type. Therefore, client code, such as in Fig. 21, can be reused with any implementation.

5. FUTURE WORK: TOWARDS FLUID AOP

This paper represents a stable snapshot of an ongoing work. In this section, we want to outline the next steps we tackle. Our future work on this model heads for *fluid aspect-oriented programming*, a term recently coined by Gregor Kiczales. In [11], he writes:

"But we can also see signs of a next generation of AOP technology, that we call fluid AOP. Fluid AOP involves the ability to temporarily shift a program (or other software model) to a different structure to do some piece of work with it, and then shift it back."

We think that due to its expressiveness concerning on-demand remodularizations, our approach is a good starting point for fluid AOP. Our idea of enabling fluid AOP in our approach is based on two concepts by which we plan to extend our model: *callback methods* and *callback activation*. The rationale behind these two concept is that we feel that we need something more dynamic than pure compile time transformations with little runtime semantics as in AspectJ [12]. Fig. 23 illustrates our ideas by code for online scheduling, i.e., a new schedule is automatically computed every time an assignment of teachers or student years to courses changes.

The method `courseAssignmentChanged()` in `SchedulingGraph` is a sample callback method. A callback method describes events (or *joinpoints*) that should lead to the execution of that method. For example, the definition of `courseAssignmentChanged()` states that this method should be called after the methods `assignCourse()` or `addRequiredCourse()` have been called on objects of type `Teacher` and `StudentYear`, respectively. Callback methods as illustrated by the example resemble pretty much advices in AspectJ [12].

Despite the similarity at the syntactic level, which stems from the fact that for the purpose of illustrating the idea of how our model can be seen as a step towards fluid AOP we basically borrow the joinpoint designator syntax of AspectJ, our callback methods are different in two important ways. First, our callback methods are real methods just as any method in a Java class: They have a signature and can be inherited and refined. This is not true for AspectJ advices. Details about the signature of a callback method as well as the inheritance rules are out of the scope of this paper.

Second, the pure declaration of such a callback method does not have any computational effects. This is the point where our new more dynamic notion of aspects and joinpoints comes in: The callback methods have to be *activated* by means of an `apply` block. An example can be found in the lower part of Fig. 23. An `apply` block is configured with an object whose definition contains callback methods, such as the `sg` object in Fig. 23. The semantics of the `apply` block is that the callbacks defined in `sg` are active during the execution of everything that is inside the `apply` block - in this case, the call to `startCourseAssignment()` and everything that is transitively called in that method.

With respect to the idea of fluid AOP, the application is transformed to perform online scheduling during the execution of a method call, and after the execution, everything is shifted back.

Please note the dynamics that is involved in this process. First, we can choose at runtime whether we want to perform online scheduling or not (depending on the runtime value of the `wantOnlineScheduling` variable. Second, if we decided to use online scheduling, we can still choose from different scheduling (respectively coloring) algorithms which we want to use (exemplified by the decision between `SucAugSched` and `SimAnSched`).

The full details of this approach - for example, the exact syntax for specifying events and options for on efficient implementation - have not yet been worked out. Yet we are confident that we will be able to answer all open questions

```
class SchedulingGraph binds ColoredGraph {
  // Remainder as in Fig. 18

  Course[] courses;
  SchedulingGraph(Courses c[]) { courses = c;}

  void courseAssignmentChanged(Course c)
    after calls(Teacher.assignCourse(c)) ||
         calls(StudentYear.addRequiredCourse(c)) {
      computeMinimumColoring(courses);
  }
}

...

// SucAugSched and SimAnSched as in Fig. 21

if (wantOnlineScheduling) {
  SchedulingGraph sg = wantSucAug ?
    new SucAugSched(courses) : new SimAnSched(courses);

  apply (sg) in {
    startCourseAssignment(courses, teachers, studentyears);
  }
} else {
    startCourseAssignment(courses, teachers, studentyears);
}
```

Figure 23: Fluid AOP with On-Demand Remodularization, Callbacks, and Callback Activation

in the next time and that the prospects of this approach are worth the trouble.

6. RELATED WORK

Pluggable Composite Adapters (PCAs) [21] and their predecessor, *Adaptive Plug and Play Components (APPCs)* [20], have been important starting points for our work. Both approaches offer different means for on-demand remodularization. The APPC model had a vague definition of required and provided interfaces. However, this feature was rather ad-hoc and not well integrated with the type system. Recognizing that the specification of the required and expected interfaces of components was rather ad-hoc in APPCs, PCAs even dropped this notion and reduced the declaration of the expected interface to a set of standard abstract methods. With the notion of collaboration interfaces, the approach presented here represents a qualitative improvement over PCA and APPC.

Due to the lack of decoupling of the component implementations from their bindings, the connectors and adapters in APPC and PCA models are bound to a fixed component. Furthermore, the lack of the notion of virtual types is another drawback of these approaches as compared to the work presented here. In addition, both approaches rely on a dedicated mapping sublanguage that is less powerful than our notion of object-oriented wrappers with wrapper recycling. Among these approaches, the APPC model of remodularization is a class-based one, and only PCAs share the object-based on-demand remodularization with our approach.

In [7], a variant of the PCA construct, called *dynamic view connectors (DVCs)* is used in the architecture of an integrated software engineering environment to support the late integration of independently developed software engineering tools. This work demonstrates the power of on-demand remodularization in a real-world, fairly large system. By being basically a realization of the PCA concept, DVCs also share their shortcomings mentioned above.

The Hyperspaces model and its instantiation in Hyper/J [27] also support on-demand remodularization - this notion was actually first introduced by the Hyperspaces model. Both, on-demand remodularization in Hyper/J and in our approach, have a common goal: "On-demand remodularization allows a developer to choose at any time the best modularization, based on any or all of their concerns, for the development task at hand" [22]. Hence the same name.

However, despite the common goal, there are some important differences between these two approaches. In a nutshell, the functionality offered by Hyper/J can be summarized as *extracting concerns* and *composing concerns*.

Extracting concerns means that one can take a piece of existing software and tag parts of the software, e.g., method `a()` in class `A` and method `b()` in class `B`, by means of a so-called *concern mapping*. Later, this mapping can be used to extract a particular concern from this software and reuse it in a different context. This is similar to the old idea of retroactive generalization in inheritance hierarchies [24]. An important concept for extracting concerns is the notion of *declarative completeness*. Basically, this means that all methods that are used inside the tagged methods but are not tagged themselves are declared as *abstract* in the context of the extracted concern. Our model does not have any dedicated means for feature extraction.

However, we think that with respect to *composing concerns* our approach is in some important ways superior to Hyper/J. Composition in Hyper/J happens by means of a so-called *hypermodule specification*, which describes in a declarative sublanguage, how different concerns should be composed. In terms of our model, a hypermodule performs both the functionality of our binding classes and the actual composition with the `+` operator. Due to this mixing and due to the absense of an interface concept similar to our collaboration interface, Hyper/J has no polymorphism and reuse as in our approach, e.g., one cannot switch between different implementations and bindings, and one cannot use them polymorphically. Since the mapping sublanguage is declarative, it relies on similar signatures that can be mapped to each other, and transformations other than name transformations (e.g., type transformations), are very difficult. In addition, Hyper/J's sublanguage for mapping specifications from different hyperslices is fairly complex and not well integrated into the common OO framework.

The last important difference is that Hyper/J's approach is class-based: it is not possible to add the functionality defined in a hyperslice to individual objects, instead the objects have to be created as objects of the compound hypermodule from the very beginning. Therefore, multiple independent bindings that are added to individual objects at runtime are not possible.

At this point, the question rises of how to position the work presented here with respect to previously published works on *collaboration-based decomposition* (CBD). CBD approaches aim at providing modules that encapsulate a whole collaboration of classes. With CBD classes are decomposed into the roles they play in the different collaborations. The idea is nicely visualized by a two dimensional matrix with the classes as the column indexes and collaborations in which these classes are involved as the row indexes.

Mixin Layers [26] and delegation layers [23] are two representatives of approaches to CBD. Both approaches provide concepts for composing and decomposing a collaboration into *layers*, such that a particular collaboration variant can be obtained by composing the required layers. Mixin layers use a nested variant of mixin-inheritance [4], whereas delegation layers combine delegation and virtual classes in order to defer the layer combination until runtime. None of these approaches support on-demand remodularization. The definition of a collaboration layer in these approaches also encodes how the collaboration will be integrated. The vocabulary of abstractions that are involved in an application is defined a-priori to the definition of any collaboration layer and is consequently shared by all layer definitions.

Collaboration layers are especially useful in case we have many different variants of a particular collaboration (for example, `Graph`, `ColoredGraph`, and `WeightedGraph`) and want to mix-and-match these variants at runtime (for example, create a `ColoredWeightedGraph` by composing the color and the weight layer). Collaboration layers complement the concepts proposed in this paper very nicely, because they would allow us to decompose both components and connectors into layers that could be combined on-demand. In the future, we plan to combine the dynamic composition features of the delegation layers approach with the concepts of the work presented here.

VanHilst and Notkin propose an approach for modelling collaborations based on templates and mixins as an alternative to using frameworks [30]. However, this approach may result in complex parameterizations and scalability problems. A *contract* [9] allows multiple potentially conflicting component customizations to exist in a single application. However, contracts do not allow conflicting customizations to be simultaneously active. Thus it is not possible to allow different instances of a class to follow different collaboration schemes.

Lasagne [29] is a runtime architecture that features aspect-oriented concepts. An aspect is implemented as a layer of wrappers. Aspects can be composed at run-time, enabling dynamic customization of systems, and context-sensitive selection of aspects is realized, enabling client-specific customization of systems.

Hölzle [8] analyses some problems that occur when combining independent components. Our proposal can be seen as an answer to the problems and challenges discussed in [8]. Mattson et al [17] also indicate the problems with framework composition, analyze reasons for these problems and investigate the state of the art of available solutions. In [3], Bosch proposes a language construct for specifying a class as the adapter of another class, that is, for explicit expression of the adapter pattern. The adapter construct as proposed in [3] has two main restrictions: First, it does not support adaptation of entire collaborative functionality. Second, as indicated in [3], it does not allow interface incompatibility.

Our work is also related to *architecture description languages* (ADL) [25], for example Rapide [14], Darwin [16], C2 [19], and Jiazzi [18]. The building blocks of an architectural description are components, connectors, and architectural configurations. A component is a unit of computation or data store, a connector is an architectural building block used to model interactions among components and rules that govern those interactions, and an architectural configuration is a connected graph of components and connectors that describe architectural structure. In comparison with our approach, ADLs are less integrated into the common OO framework, and do not have a dedicated notion of on-demand remodularization in order to provide a new virtual interface to a system.

We think that collaboration interfaces might also prove very useful in the context of ADL. In ADL, components also describe their functionality and dependencies in the form of required and provided methods (so-called *ports*). The goal of these ports is to render the components reusable and independent from other components. However, although the components are syntactically independent, there is a very subtle semantic coupling between the components, because a component `A` that is to be connected with a component `B` has to provide the exact counterpart interface of `B`. The situation becomes even worse if we consider multiple components that refer to the same protocol. The problem is that there is no central specification of the communication protocol to which all components that use this protocol can refer to – in other words: we have no notion of a collaboration interface.

7. SUMMARY

This paper proposed language concepts that facilitate the separation of an application into independent reusable building blocks and the integration of pre-build generic software components into applications that have been developed by third party vendors.

A key element of our approach is the notion of *collaboration interfaces*, used to declare the type of generic components. Collaboration interfaces are nested interfaces, bundling several abstractions that together build up the concept world of a component type into a family of virtual types [6]. In addition to the 'client-from-server contract', expressed by standard interfaces, collaboration interfaces also capture what servers expect from potential client contexts in which they might be integrated, i.e., the server-from-client contract. The implementations of these two contracts are completely decoupled from each other.

The implementation of the second contract translates the abstractions and vocabulary of an existing code base into the vocabulary understood by a set of components that are connected by a common collaboration interface. This translation is called *on-demand remodularization*, since the translation is virtual and effective only during the execution of functionality in the collaboration interface, whose server-from-client contract is implemented by the remodularization. Our approach to remodularization is object-based and uses the full computational power of an object-oriented language. The concept of *wrapper recycling* was additionally introduced to support the specification of the remodularization.

The decoupling of component implementation from bindings via remodularizations, allows to mix-and-match remodularizations and components on demand. The `+` operator was introduced for this purpose. This decoupling combined with the lean integration of collaboration interfaces with generalized notions of inheritance and subtype polymorphism, provide for a high degree of reuse in our model.

There are several areas of future work which were briefly outlined in the paper. First, we plan to combine our approach with concepts from the delegation layers approach [23]. This will allow us to decompose both components and connectors into layers that could be flexibly combined on-demand. Another very important area of future work is to extend our remodularization language with constructs for better supporting the integration of reusable crosscutting concerns, as outlined in Sec. 5. This is a promising step towards fluid AOP.

8. REFERENCES

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Programming*. Springer, 1993.

[2] L. M. Berlin. When objects collide: Experiences with reusing multiple class hierarchies. In *Proceedings of OOPSLA '90*, pages 181–193, 1990.

[3] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices 25(10)*, pages 303–311, 1990.

[5] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[6] E. Ernst. Family polymorphism. In *Proceedings of ECOOP '01*, LNCS 2072, pages 303–326. Springer, 2001.

[7] S. Herrmann and M. Mezini. PIROL: A case study for multidimensional separation of concerns in software engineering environments. In *Proceedings of OOPSLA 2000.* ACM SIGPLAN Notices, 2000.

[8] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings ECOOP '93, LNCS*, 1993.

[9] I. M. Holland. Specifying reusable components using contracts. In *Proceedings ECOOP '93, LNCS 615*, pages 287–308, 1992.

[10] Java Foundation Classes. http://java.sun.com/products/jfc/.

[11] G. Kiczales. Aspect-oriented programming - the fun has just begun. In *Workshop on New Visions for Software Design and Productivity: Research and Applications*, Vanderbilt University, Nashville, Tennessee, December 13-14, 2001.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyvaskyla, Finland, 1997. Springer-Verlag.

[14] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[15] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89.* ACM SIGPLAN, 1989.

[16] J. Magee and J. Kramer. Dynamic structure in software architecture. In *Proceedings of the ACM SIGSOFT'96 Symposium on Foundations of Software Engineering*, 1996.

[17] M. Mattson, J. Bosch, and M. E. Fayad. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10), October 1999.

[18] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned java. In *Proceedings of OOPSLA '01*, 2001.

[19] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of off-the-shelf components in C2-style architectures. In *Proceedings of the 1997 international conference on Software engineering*, pages 692–700, 1997.

[20] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, 1998.

[21] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer, 2001. University of Twente, The Netherlands.

[22] H. Ossher and P. Tarr. On the need for on-demand remodularization. In *ECOOP'2000 workshop on Aspects and Separation of Concerns*, 2000.

[23] K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of ECOOP '02, LNCS 2374, Springer*, 2002.

[24] C. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *OOPSLA '89 Proceedings*, 1989.

[25] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* PrenticeHall, 1996.

[26] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *Proceedings of ECOOP '98*, pages 550–570, 1998.

[27] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. International Conference on Software Engineering (ICSE 99)*, 1999.

[28] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97*, 1997.

[29] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Joergensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, 2001.

[30] M. VanHilst and D. Notkin. Using role components to implement collaboration-based design. In *Proceedings OOPSLA 96*, 1996.